C++学习笔记9

2022年3月19日 20:57

- 动态内存和智能指针
  - 智能指针
    - 为了更加容易和安全的使用智能指针，新的标准库提供了两种智能指针类型来管理对象
    - share-ptr：允许多个指针指向同一个对象
    - unique-ptr：独占所指向的对象
    - weak-ptr：这是一个伴随类，他是一个弱引用，指向share-ptr所管理的对象
    - 这三种类型的指针都定义在memory中
  - share-ptr
    - 智能指针使用方式与普通模板类似，都需要额外的类型信息
      share-ptr《string》 a（指向string的指针a）
    - share-ptr和unique-ptr支持的操作
      - 表 12.1：shared_ptr 和 unique_ptr 都支持的操作

shared_ptr<T> sp unique_ptr<T> up	空智能指针，可以指向类型为 T 的对象
p	将 p 用作一个条件判断，若 p 指向一个对象，则为 true
*p	解引用 p，获得它指向的对象
p->mem	等价于 (*p).mem
p.get()	返回 p 中保存的指针。要小心使用，若智能指针释放了其对象，返回的指针所指向的对象也就消失了
swap(p, q) p.swap(q)	交换 p 和 q 中的指针

      - 表 12.2：shared_ptr 独有的操作

make_shared<T>(*args*)	返回一个 shared_ptr，指向一个动态分配的类型为 T 的对象。使用 *args* 初始化此对象
shared_ptr<T>p(q)	p 是 shared_ptr q 的拷贝；此操作会递增 q 中的计数器。q 中的指针必须能转换为 T*（参见 4.11.2 节，第 143 页）
p = q	p 和 q 都是 shared_ptr，所保存的指针必须能相互转换。此操作会递减 p 的引用计数，递增 q 的引用计数；若 p 的引用计数变为 0，则将其管理的原内存释放
p.unique()	若 p.use_count() 为 1，返回 true；否则返回 false
p.use_count()	返回与 p 共享对象的智能指针数量；可能很慢，主要用于调试

    - 最安全的还是使用make-shared分配空间
      - make-shared《int》（xx）
      - 也采用模板的方式，但是初始化需要用（）
      - 当没有share-ptr指向原先用make-shared创建对象，原先创建的对象会被自动释放内存
      - 对于一个普通指针，在一个作用域内部创建，在程序结束的时候不会消除，对于智能指针会
      - 那个函数创建的智能指针就是那个函数的指针，在出自己函数的时候会被自己销毁，传到其他函数时候不受影响
      - 如果将shared-ptr存放在一个容器中，而后不需要全部元素，而只是使用其

中有一部分，要记得用erase删除不再需要的哪些元素
        - 对于有多个shared-ptr的指针，只要在最后一个指针没有销毁之前不会被释放
- new的使用
    - 我们可以使用new分配空间。
        - new定义在new文件中
        - new的操作
            - int *pi=new int----》pi指向指向一个动态分配的未初始化的对象
                - 默认情况下，动态分配的对象（vector，string等）是默认初始化的，但是int等对象的值是没有定义的
            - 初始化
                - 我们可以使用原先的初始化方式和构造方式分配对象
                  int *p=new string（10，‘9’）；。。。
                    - 对于使用圆括号且圆括号只有单一的初始化器 的时候可以使用auto进行推断
                      auto p=new string（obj）；
                - 我们直接在后面加上一个空括号表示对new的对象进行值初始化
                  int *pi=new int（）： *pi为0
            - const
                - new可以分配const对象，但是必须要初始化，对于类要有隐式初始化
                  const int *pi=new const int（2022）；
            - 内存耗尽
                - 在内存耗尽的时候，使用new分配内存会报bad-alloc，可以改变使用new的方式
                    - 在new后面更一个括号，阻止他抛出异常
                      int *p=new（nothrow）int//如果分配失败，new返回一个空指针
            - 释放动态内存
                - 为了防止内存耗尽在动态内存使用完成后，必须将其归还给系统，delete表达式接受一个指针指向我们想要释放的对象
                  delete p//p必须是一个指向动态分配的对象或是一个空指针
                - 我们不能释放一块非new分配的内存，以及将一块内存同时释放多次
                - 内存释放的同时指针失效
                - delete可以释放使用new分配的const对象
                - 由内置指针管理的动态内存在被显示释放前一直都会存在
            - 注意
                - 对于初始化：出于变量初始化相同的原因，对动态分配的对象进行初始化通常是个好主意

        - ◇ 动态内存的管理非常容易出错：
            - ▸ 忘记delete内存
            - ▸ 使用已经释放掉了的对象
            - ▸ 同一块内存释放两次
            - ▸ 建议：坚持只使用智能指针，就可以避免所有这些问题
- share-ptr和new的结合使用
    - 我们可以使用new的指针直接初始化智能指针，但是不能值初始化，因为指针的构造函数时explicit
        - share-ptr 《int》 p（new int （42））p是一个指向int的智能指针，*p的值是42·
    - 出于相同的原因，使用new初始化智能指针是不行的
    - share-ptr和new混合使用的方法

表 12.3：定义和改变 shared_ptr 的其他方法

shared_ptr<T> p(q)	p 管理内置指针 q 所指向的对象；q 必须指向 new 分配的内存，且能够转换为 T*类型
shared_ptr<T> p(u)	p 从 unique_ptr u 那里接管了对象的所有权；将 u 置为空
shared_ptr<T> p(q, d)	p 接管了内置指针 q 所指向的对象的所有权。q 必须能转换为 T*类型（参见 4.11.2 节，第 143 页）。p 将使用可调用对象 d（参见 10.3.2 节，第 346 页）来代替 delete

shared_ptr<T> p(p2, d)	如表 12.2 所示，p 是 shared_ptr p2 的拷贝，唯一的区别是 p 将用可调用对象 d 来代替 delete
p.reset() p.reset(q) p.reset(q, d)	若 p 是唯一指向其对象的 shared_ptr，reset 会释放此对象。若传递了可选的参数内置指针 q，会令 p 指向 q，否则会将 p 置为空。若还传递了参数 d，将会调用 d 而不是 delete 来释放 q

        - 注意
            - ◆ 当一个share-ptr绑定到一个普通指针上时候，我们就将内存管理任务交给share-ptr了，一旦这样做了，我们就不应该再使用内置指针来访问share-ptr所指向的内存了。
            - ◆ 使用一个内置指针来访问一个智能指针所负责的对象是很危险的，因为我们无法直到对象何时被销毁
            - ◆ 不要混合使用普通指针和智能指针，也不要使用get初始化另一个智能指针或为智能指针赋值
            - ◆ get用来将指针的访问权限传递给代码，你只有在确定代码不会delete指针的情况下才能使用get。
            - ◆ 永远不要用get初始化另一个智能指针或者为另一个智能指针赋值
            - ◆ 不使用相同的内置指针初始化（或reset）多个指针。
            - ◆ 不delete get（）返回的指针
            - ◆ 不使用get（）初始化或reset另一个智能指针
    - 其他的share-ptr操作
        -
- unique-ptr
    - 一个unique-ptr拥有它所指向的对象，与share-ptr不同，某个时刻只能有一个unique-ptr指向一个给定的对象

- 当unique-ptr被销毁时，他所指的对象也会被销毁
- 操作
  - □ 。

表 12.4：unique_ptr 操作（另参见表 12.1，第 401 页）	
unique_ptr<T> u1	空 unique_ptr，可以指向类型为 T 的对象。u1 会使用 delete
unique_ptr<T, D> u2	来释放它的指针；u2 会使用一个类型为 D 的可调用对象来释放它的指针
unique_ptr<T, D> u(d)	空 unique_ptr，指向类型为 T 的对象，用类型为 D 的对象 d 代替 delete
u = nullptr	释放 u 指向的对象，将 u 置为空
u.release()	u 放弃对指针的控制权，返回指针，并将 u 置为空
u.reset()	释放 u 指向的对象
u.reset(q)	如果提供了内置指针 q，令 u 指向这个对象；否则将 u 置为空
u.reset(nullptr)	

- 当我们要定义一个unique-ptr时，需要将其绑定的到一个new返回的指针上。类似于shared-ptr，初始化unique-ptr必须采用直接初始化的形式
  unique-ptr《double》p1（new int（42））
- 对于拷贝于赋值，我们可以拷贝或者赋值一个将要被销毁的unique-ptr。
  - □ 可以从函数中返回一个unique-ptr
  - □ 还可以返回一个局部对象的拷贝
- 对于其的删除操作，与share-ptr不同
  - □ share-ptr需要在括号里面提供一个用于删除的函数
    share-ptr<int> p(&c,deleter)
  - □ unique-ptr还需要再类型里面再提供一次
    unique-ptr<int,decltype(deleter)*> p(&c,deleter);

○ weak-ptr
- weak-ptr是一种不控制所指向对象生存期的智能指针，它指向一个由一个shared-ptr管理的对象。
- 将一个weak-ptr绑定到一个shared-ptr不会改变shared-ptr的引用计数
- 一旦最后一个指向对象的shared-ptr被销毁，对象就会被释放。
- 即使weak-ptr指向对象，对象也还是会被释放，因此weak-ptr的名字抓住了这种智能指针“弱”共享对象的特点
- 操作

□

表 12.5：weak_ptr	
weak_ptr<T> w	空 weak_ptr 可以指向类型为 T 的对象
weak_ptr<T> w(sp)	与 shared_ptr sp 指向相同对象的 weak_ptr。T 必须能转换为 sp 指向的类型
w = p	p 可以是一个 shared_ptr 或一个 weak_ptr。赋值后 w 与 p 共享对象
w.reset()	将 w 置为空
w.use_count()	与 w 共享对象的 shared_ptr 的数量
w.expired()	若 w.use_count() 为 0，返回 true，否则返回 false
w.lock()	如果 expired 为 true，返回一个空 shared_ptr；否则返回一个指向 w 的对象的 shared_ptr

- 创建weak-ptr
  - □ 我们需要使用一个share-ptr来初始化它
    share-ptr《int》 p=make-shared《int》（）
    weak-ptr《int》wp（p）

        - 本例中wp和p指向相同的对象，由于是 弱共享，创建wp不会改变p的引用计数，wp指向的对象可能被释放掉
        - 我们不能使用weak-ptr直接访问对象，而必须调用lock
    - 注意
        - 对于指针的初始化一定需要指针进行
- 动态数组
    - new和数组
        - 为了能让new分配一个对象数组，我们要在类型名之后跟一对方括号，在其中指明要分配的对象和数目
          int *p=new int【12】
        - 我们不能会分配出来的内存调用begin，end，不能用范围for处理元素
        - 动态数组不是数组类型
        - 初始化分配对象的数组
            - 默认状态下，new分配的对象，不管是单个分配的还是数组中的，都是默认初始化的。可以对数组中的元素进行值初始化。
                - 方法是在大小后面跟一对括号
                  int *o=new int【10】（）
                - 也可以使用花括号指定
                  int *p=new int【10】{1，2，3，4，5，6，7，8，9，0}
            - 动态分配一个空数组是合法的，但是不能解引用
        - 删除数组
            - 对于一个数组，我们使用一种特殊形式，delete：delete 【】 pa
            - 数组中元素按逆序销毁，即最后一个元素首先被销毁，然后是到数第二个，依次推类
        - 指向数组的unique-ptr
            - 可以定义一个指向数组的unique-ptr，定义后操作会不一样
                - 定义：unique-ptr《int【】》 up
            - 操作
                - 表 12.6：指向数组的 unique_ptr

指向数组的 unique_ptr 不支持成员访问运算符（点和箭头运算符）。	
其他 unique_ptr 操作不变。	
unique_ptr<T[]> u	u 可以指向一个动态分配的数组，数组元素类型为 T
unique_ptr<T[]> u(p)	u 指向内置指针 p 所指向的动态分配的数组。p 必须能转换为类型 T*（参见 4.11.2 节，第 143 页）
u[i]	返回 u 拥有的数组中位置 i 处的对象 u 必须指向一个数组

            - 与unique-ptr不同，share-ptr
                - 如果要支持管理动态数组，必须同时定义一个删除器，这个删除器是一个函数
                - share-ptr不支持下标操作，必须要使用指针进行操作数组中的元素
- allocator类
    - 标准库allocator类定义在文件memory中，他帮助我们将内存分配和对象构造分离开

- 它提供一种内存感知的方法，它分配的内存是原始的、未构造的。
- allocator定义的操作
  - allocator类及其算法

表 12.7：标准库 allocator 类及其算法

`allocator<T> a`	定义了一个名为 a 的 allocator 对象，它可以为类型为 T 的对象分配内存
`a.allocate(n)`	分配一段原始的、未构造的内存，保存 n 个类型为 T 的对象
`a.deallocate(p, n)`	释放从 T*指针 p 中地址开始的内存，这块内存保存了 n 个类型为 T 的对象；p 必须是一个先前由 allocate 返回的指针，且 n 必须是 p 创建时所要求的大小。在调用 deallocate 之前，用户必须对每个在这块内存中创建的对象调用 destroy
`a.construct(p, args)`	p 必须是一个类型为 T*的指针，指向一块原始内存；*arg* 被传递给类型为 T 的构造函数，用来在 p 指向的内存中构造一个对象
`a.destroy(p)`	p 为 T*类型的指针，此算法对 p 指向的对象执行析构函数（参见 12.1.1 节，第 402 页）

- alloctaor分配，构造，与删除
  - 使用allocator可以直接分配一段原始的内存，在需要的时候使用allocate函数对象的分配（但不初始化），返回一个指针，指向完成之后的内存的起始点

```
allocator<string> alloc;              // 可以分配 string 的 allocator 对象
auto const p = alloc.allocate(n);     // 分配 n 个未初始化的 string
```

  - 可以使用construct构造对象
    - construct接受一个指针和额外的参数
      - 其中一个指针是指向原来已经分配但是没有构造对象的内存的
      - 额外参数是用来初始化构造对象的
      - 例如：我们可以从第一个已经分配好的内存开始，向里面进行初始化后让指针移动到下一个分配的内存中

```
auto q = p; // q 指向最后构造的元素之后的位置
alloc.construct(q++);              // *q 为空字符串
alloc.construct(q++, 10, 'c');     // *q 为 cccccccccc
alloc.construct(q++, "hi");        // *q 为 hi!
```

    - 对未构造对象的原始内存解引用是错误的。使用未构造的内存，结果是未定义的
  - 删除元素
    - 我们使用destroy进行销毁原先构造的对象
    - 对于对象，destroy只能对已经构造的对象一个一个的销毁
    - 我们只能对真正构造了元素的对象进行destroy

```
while (q != p)
    alloc.destroy(--q); // 释放我们真正构造的 string
```

  - 释放内存
    - 在元素被销毁后，我们可以重新construct（注意指针），也可以将内存归还给系统，调用deallocate
      alloc.deallocate(p,n)
    - 其中指针不能为空必须指向由allocate分配的内存，传递的值需要和分配的一样
- allocator定义的算法
  - allocator定义了两个伴随算法，定义在memory里
  - 算法

- 这些算法只对allocator有用

表 12.8：allocator 算法

这些函数在给定目的位置创建元素，而不是由系统分配内存给它们。

`uninitialized_copy(b,e,b2)`	从迭代器 b 和 e 指出的输入范围中拷贝元素到迭代器 b2 指定的未构造的原始内存中。b2 指向的内存必须足够大，能容纳输入序列中元素的拷贝
`uninitialized_copy_n(b,n,b2)`	从迭代器 b 指向的元素开始，拷贝 n 个元素到 b2 开始的内存中
`uninitialized_fill(b,e,t)`	在迭代器 b 和 e 指定的原始内存范围中创建对象，对象的值均为 t 的拷贝
`uninitialized_fill_n(b,n,t)`	从迭代器 b 指向的内存地址开始创建 n 个对象。b 必须指向足够大的未构造的原始内存，能够容纳给定数量的对象

- 类似于拷贝算法，通过传入迭代器和特殊的值进行操作